

Activation Inheritance in Modelica

Ramine Nikoukhah

INRIA, BP 105, 78153 Le Chesnay, France

ramine.nikoukhah@inria.fr

Abstract

Modelica specifies two types of equations: the equations defined directly in the "equation" section, which are supposed to hold all the time, and the equations defined within a "when" statement. The latter are "activated" by explicit events at corresponding times. In making the analogy with Scicos, the equations of the first type are the counterpart of "always active" blocks whereas the second type equations are "event activated" blocks. A useful feature in Scicos is the mechanism of activation inheritance. In this paper, we examine the possible extension of the Modelica specification to introduce a similar mechanism in Modelica.

Keywords Modelica, Synchronous languages, Scicos, modeling and simulation

1. Introduction

Modelica (www.modelica.org) is a language for modeling physical systems. It has been originally developed for modeling systems obtained from the interconnection of components from different disciplines such as electrical circuits, hydraulic and thermodynamics systems, etc. These components are represented symbolically in the language providing the compiler the ability to perform symbolic manipulations on the resulting system of differential equations. But Modelica is not limited to modeling continuous-time systems; it can be used to construct hybrid systems, i.e., systems in which continuous-time and discrete-time components interact [1]. Modelica specification [2] defines the way these interactions should be interpreted and does so by inspiring from the formalism of synchronous languages. Modelica hybrid formalism can be interpreted or extended to include many features available already in Scicos. We have examined some of these issues in previous papers [4][5]. We consider here the implementation of the activation inheritance mechanism of Scicos in Modelica.

Scicos (www.scicos.org) is a modeling and simulation

environment for hybrid systems. Scicos formalism is based on the extension of synchronous languages, in particular Signal [3], to the hybrid environment. The class of models that Scicos is designed for is almost the same as that of Modelica. So it is not a surprise that Modelica and Scicos have many similar features and confront similar problems. Just as in Modelica, Scicos is event driven, to be more precise, activation driven, i.e., the activation of equations are explicitly specified. This is in opposition to data-flow formalism where the activation is implicitly derived from the flow of data.

The data-flow mechanism has however many advantages in certain situations. That is why Scicos includes an activation inheritance mechanism providing a data-flow-like behavior without perturbing the overall formalism. The activation inheritance is treated at an early compilation phase.

In this paper, we examine the extension of the Modelica specification to implement an inheritance mechanism similar to the one available in Scicos. We work here only with flat Modelica models (usually obtained from the application of a front-end compiler).

This extension can have many applications. In the last section, we consider its possible use in the problem of code generation for control applications.

2. Basic Idea

There is a clear distinction between equations activated by inheritance and those that are declared "always active" in Scicos. This is not the case in Modelica where both types are placed in the `equation` section outside of `when` clauses. To distinguish the two types of activations, we propose a new `when` clause in this section. We start by considering only discrete dynamics.

Consider the following valid Modelica code:

```
discrete Real z;  
Real u, y;  
equation  
  u=2*y;  
  y=pre(z);  
  when sample(0,1) then  
    z=pre(z)+u;  
  end when;
```

and compare it to

```
discrete Real u, y, z;
```

```

equation
  when sample(0,1) then
    u=2*y;
    y=pre(z);
    z=pre(z)+u;
  end when;

```

Both codes are correct and should yield identical simulation codes on any reasonable compiler/simulator. In our proposal, the two codes are considered semantically equivalent. The former being transformed to the latter in a first phase of compilation by the compiler.

To see where the activation inheritance comes into play in this example, note that the equations defining u , and y are not “explicitly activated”; we say then that these equations inherit their activations. Such an equation is activated only if at least one of the variables on which it depends may potentially change value. This of course is very similar to a data-flow behavior. In this particular example, the equation defining u depended on z , that is why it was moved to the `when` clause. The equation defining y depended on u , now in the `when` clause, so it was also moved inside the `when` clause.

This may seem like a minor issue however it allows us to present useful extensions as we shall see later.

3. Presence of Discrete States

To show that the activation inheritance mechanism cannot be assimilated with code optimization, consider the following code:

```

equation
  u=2*y;
  y=pre(z);
  j=pre(j)+y;
  when sample(0,1) then
    z=pre(z)+u;
  end when;

```

This code is not currently legal in Modelica whether j is defined discrete or not. We are not going to worry about declarations as discrete; from our point of view, this is redundant information that can only be useful for checking the model. Indeed, the placement of the variable in the equation section is enough to determine its discrete nature.

Now consider the effect of the transformation we had proposed on the previous example on this new one:

```

equation
  when sample(0,1) then
    u=2*y;
    y=pre(z);
    j=pre(j)+y;
    z=pre(z)+u;
  end when;

```

This is a valid Modelica code. As we propose that the pre and post versions be considered semantically equivalent, then the first version should be considered valid as well. Note that in this example we do not merely try to propose a code optimization strategy. Unlike memory-less equations where too much activation does not affect the

simulation outcome, the activation instants of the equation $j=\text{pre}(j)+y$; must be uniquely specified in order to obtain an unambiguous model. Clearly without a rigorous definition of the concept of activation inheritance, the extension that we propose cannot work.

4. Multiple Inheritance

Let us now examine the situation where an equation depends on more than one variable updated in different `when` clauses:

```

equation
  a=k+m;
  when sample(0,1) then
    k=pre(k)+1;
  end when;
  when sample(.5,1) then
    m=pre(m)-1;
  end when;

```

In this case, the variable a can potentially change value at both `sample(0,1)` and `sample(.5,1)`. So the transformation is carried out as follows:

```

equation
  when sample(0,1) then
    k=pre(k)+1;
    a=k+m;
  end when;
  when sample(.5,1) then
    m=pre(m)-1;
    a=k+m;
  end when;

```

This is an easy case because the two `when` clauses are primary (see [5]), i.e. asynchronous. The situation is a bit more complicated when this is not the case. Consider the following example:

```

equation
  z=pre(z)+a+b;
  when sample(0,1) then
    a=pre(a)+1;
  end when;
  when a>3 then
    b=a;
  end when;

```

Clearly, we would not obtain a correct model if we placed the definition of z in both `when` clauses. In particular z would be incremented wrongly twice by $a+b$ when a crosses 3. The fundamental reason is that in this case the two `when` clauses are not asynchronous. The activations updating a and b when a crosses 3 are in fact synchronous (same activation). In [5], we showed how the secondary `when` clauses are removed in the process of compilation. Adding the inheritance mechanism, in our case this yields the following code:

```

when sample(0,1) then
  a=pre(a)+1;
  if edge(a>3) then
    b=a;

```

```

    end if ;
    z=pre(z)+a+b;
end when;
```

The situation is of course more complex in the presence of multiple primary and secondary when clauses but the inheritance rules are clearly defined and the compiler can treat them in an unambiguous manner.

5. Always Active in Modelica

So far we have considered only cases where the model contained no continuous-time dynamics. In general, the main component of a Modelica model runs in continuous-time and the expressions within an `equation` section, but outside any `when` clause, have been associated systematically to the “always active” components in Scicos. It is for this reason that in our Modelica compiler, we replicate all these expressions inside the body of all `when` statements. For example in the following example:

```

equation
  a=f(k+3);
  b=sin(time);
  when sample(0,1) then
    k=pre(k)+1;
  end when;
  when sample(.5,1) then
    m=pre(m)-1;
  end when;
```

we obtain

```

equation
  when continuous then
    a=f(k+3);
    b=sin(time);
  end when;
  when sample(0,1) then
    k=pre(k)+1;
    a=f(k+3);
    b=sin(time);
  end when;
  when sample(.5,1) then
    a=f(k+3);
    b=sin(time);
    m=pre(m)-1;
  end when;
```

This represents of course the result of the brut force application of the systematic transformations; the final code can be much smaller after optimization. The `when continuous` clause corresponds essentially to the pure continuous-time dynamics, the part that would be left to the DAE solver, see [5] for more details.

This treatment is not really consistent with the inheritance mechanism we are proposing here. If we strictly apply the inheritance mechanism described previously, there wouldn't be any `continuous` clause and the expression `a=f(k+3)` would appear only in the first `when` clause, as it should, but `b=sin(time)`; appears nowhere! Clearly there is a distinction between the two equations. The inheritance mechanism can work only if somehow we specify that `b=sin(time)`; is

always active. This can be done by introducing a new `when` clause:

```

equation
  a=f(k+3);
  when always_active then
    b=sin(time);
  end when;
  when sample(0,1) then
    k=pre(k)+1;
  end when;
  when sample(.5,1) then
    m=pre(m)-1;
  end when;
```

This way we explicitly specify that `b=sin(time)`; should be evaluated all the time, but the activation of `a=f(k+3)`; is inherited. In particular for discrete variables, the compiler replicates this expression only in `when` statements in which “k” is computed (appears on the left hand side of an equation):

```

equation
  when always_active then
    b=sin(time);
  end when;
  when sample(0,1) then
    k=pre(k)+1;
    a=f(k+3);
  end when;
  when sample(.5,1) then
    m=pre(m)-1;
  end when;
```

The equations in the always active section are then copied in all `when` clauses. Of course if we had `a=f(k+m)`; then the expression would be copied in both discrete `when` statements above (multiple inheritance).

The use of the `always_active` clause corresponds exactly to the block property “always active” in Scicos. To require such a clause however would break backward compatibility in Modelica. A possible solution would be to note that in almost all cases, always activation, which is associated with variables evolving continuously in time, originates from the use of the built-in variable `time` or the operator `der()`. It is then possible to implement a pre-compiler filter that scans over the flat Modelica model and place the equations in which `der()` and `time` appear inside an `always_active` clause. The rest of the continuous variables follow naturally by applying the inheritance rule.

6. Applications

There are three closely related issues that need to be carefully studied in Modelica: separate compilation of a part of a model, code generation for controller applications and general external functions allowing for internal states. The heart of all three problems is the ability to isolate any part of a diagram, and under certain conditions, generate a C code providing the overall behavior of this part with a canonical API.

The simplest situation is when this part of the diagram contains only memory-less event-less dynamics in which

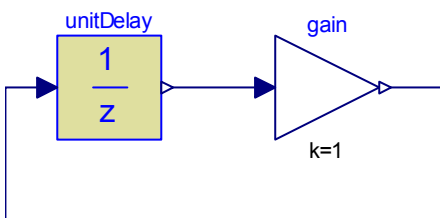
case the C code would be an external function as it already exists in Modelica.

The situation becomes more interesting when the part of the diagram we are considering contains discrete states. Consider for example a classical controller configuration where a continuous-time plant is controlled by a discrete-time, single frequency controller. By isolating the controller part, we end up with models driven synchronously. Based on the current interpretation of the Modelica specification by Dymola, these discrete blocks are each driven by a `sample` event generator; the use of an identical period and phase in the sample statements provides the synchronization. We have tried to show in [5] that this form of implicit synchronization detected only during simulation puts useless constraints on the model and the compiler, and it should be avoided. An alternative solution was proposed in [6] where the keyword `sample` was given a macro status allowing for proper synchronization via a clock calculus similar to that used in Simulink and `SampleCLK` blocks in Scicos.

Another technique to synchronize the discrete blocks would be the explicit use of events as signals. For that, we introduced a new type called `Event` in [4]. In this approach, every discrete block would have an event input port (called activation input port in Scicos) driving its activations. This way, a single `sample` event generator can be hooked up to all discrete blocks driving them synchronously.

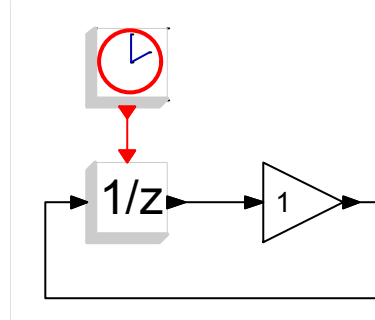
The inheritance mechanism provides an alternative solution to the synchronization problem. In most cases, there are already blocks in the discrete part that work almost on the inheritance mechanism. For example a simple `Gain` block used in this part is not driven by any event: the corresponding equations are placed in the equation section. We have seen that this is pretty much an inheritance mechanism relying on compiler optimization. With the inheritance mechanism we have introduced, models with states such as discrete-state space blocks can also be driven via inheritance. In fact it almost suffices to remove the `when`, `end when` statements from the existing models. In an actual implementation, it suffices to drive a first block (usually the analog to digital converter block defining the boundary between plant and controller) with the proper `sample` event generator. The rest of the blocks then follow through by the inheritance mechanism synchronously as they should. The advantage of the inheritance solution is that the discrete part need not even run on a fixed frequency clock.

Let us examine these ideas on actual examples. In the following Modelica diagram, we have a purely discrete single frequency system.



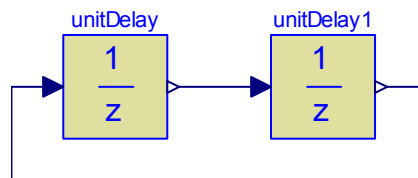
The `Memory` block is driven by a `sample` event generator (equations are all within a `when` clause) but the `Gain` block contributes an equation to the equation section, outside the `when` clause. The inheritance mechanism would move the `Gain` equation into the `when` clause yielding a purely discrete system (a system where all variables are discrete).

The Scicos counterpart of this Modelica model is

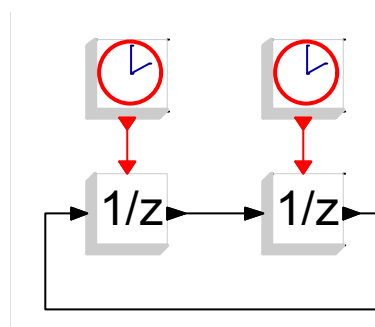


where the only difference is the explicit presence of an event clock (generator). Note that the event clock needs not have fixed frequency in this case.

Consider now the case where there are two discrete blocks in a diagram:

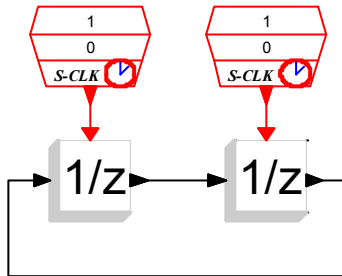


If each block has its own `sample` event generator, as it is the case today in the discrete Modelica library, the Scicos counterpart would be

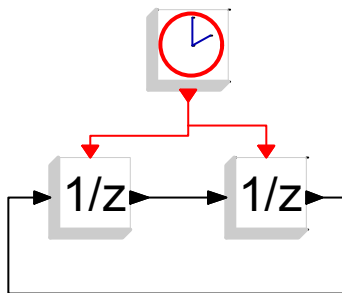


This diagram is not synchronous in Scicos even if the two event generators have identical periods. The simulation result for this diagram would not be predictable in Scicos

and we think it should not be in Modelica either unless a new definition is associated with the keyword `sample` as we have proposed in [6] where the keyword `sample` was given a macro status allowing for proper synchronization via a clock calculus similar to the one used in Simulink and SampleCLK blocks in Scicos. The corresponding Scicos diagram using SampleCLK blocks would be the following:



In this case, the Scicos compiler starts by performing the necessary computations on SampleCLK blocks' periods and offsets to find the unique slowest clock that by subsampling can replace the activations of these blocks. In this case the computation is trivial because the two blocks have identical periods; the equivalent diagram after this first phase is the following:



By adopting the proposed interpretation for the Modelica keyword `sample` in [6], Modelica compiler would function similarly.

In Scicos, it is also possible to construct directly this latter diagram, i.e., to drive both memory blocks by the same event block. This is not possible to do in Modelica unless we adopt the Event type suggested in [4]. In that case we would be able to express the Memory block as follows:

```

model Memory
  input Event e1;
  output Real y;
  input Real u;
  discrete Real z;
  equation
    when e1 then
      z=u;
      y=pre(z);
    end when;
end Memory;

```

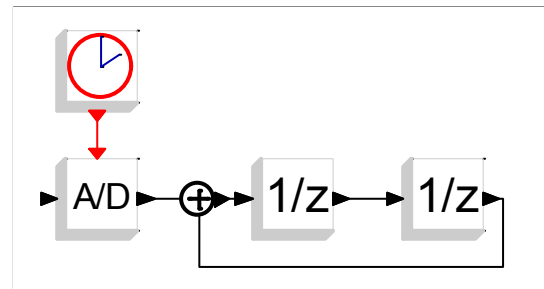
We would also need an event generator block, something resembling the following:

```

Event e(start=t0) ;
discrete Integer k(start=0) ;
equation
  when pre(e) then
    k=pre(k)+1 ;
    e=k*T+t0 ;
  end when ;

```

In Scicos, a Memory block, like any other block, can also be defined without an activation input port in which case it inherits its activation from its regular input. To see how this works, consider a slightly more complicated diagram that corresponds to a simple controller part of a Plant/controller diagram:



In this case the Memory blocks (and the sum block as well) are activated by inheritance. The Modelica counterpart of this Memory block would be:

```

model Memory
  output Real y;
  input Real u;
  discrete Real z;
  equation
    z=u;
    y=pre(z);
  end Memory;

```

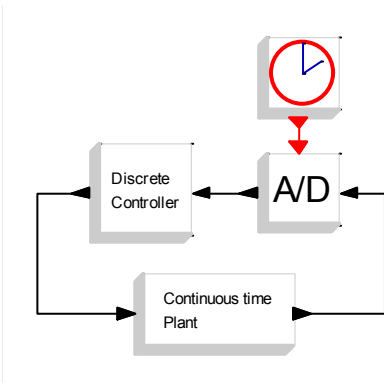
This means in particular that in a controller model, by defining all discrete blocks “inheriting”, i.e., without any built-in when `sample` clauses, it suffices to drive only the interface between the plant and the controller with an event generator. This interface would be a model such as the following:

```

model AD
  discrete output Real y;
  input Real u;
  parameter Real T=1;
  equation
    when sample(0,T) then
      y=u;
    end when;
  end AD;

```

It is the activation in the when `sample` clause of this block that is inherited by the controller part which would then run synchronously. The following is the general picture of this setup:



The continuous-time plant does not inherit any activation because it is already always active. This configuration is a classical representation used by control engineers. Note that no D/A block is necessary here because the output of discrete blocks hold their outputs constant until their next activation, nonetheless such a block can be used to clearly indicate the boundaries of the controller.

7. Conclusion

We have proposed an activation inheritance mechanism in Modelica that can be used to synchronize a set of discrete blocks, for example the controller part of a standard plant/controller configuration. This extension is one possible way to solve the synchronization issues in the discrete block library.

With this new extension, we have three different solutions to the synchronization problem: the use of macro `sample` clauses (similar to `SampleCLK` in Scicos) in different discrete blocks, the use of event input/outputs allowing the activation of multiple blocks via a single event generator, and activation inheritance. These solutions are complementary and can very well co-exist in Modelica; they do in Scicos.

The macro `sample` solution is limited to periodic systems. It is particularly useful for modeling synchronous multi-frequency systems. Having access to the `sample` parameters in each block allows also the computation of blocks parameters as a function of `sample` parameters. For example, the parameters of a discrete linear system block may be computed as a function of the sampling period using the exact discretization method.

The use of the Event type provides full control over the activation and synchronization of discrete blocks and is a key element needed for separate compilation in Modelica. It allows synchronous, asynchronous, single frequency, multi-frequency and sporadic activations.

Finally inheritance activation provides a modeling facility that avoids the need for explicitly defining the activation of each block. The use of inheriting blocks provides a viable alternative in the single frequency case to the other two solutions.

Acknowledgments

This work has been supported by the ANR project SIMPA2-C6E2.

References

- [1] M. Otter, H. Elmqvist, S. E. Mattsson, "Hybrid Modeling in Modelica based on the Synchronous Data Flow Principle", CACSD'99, Aug; 1999, Hawaii, USA.
- [2] Modelica Association, Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling, www.modelica.org/documents/ModelicaSpec30.
- [3] A. Benveniste, P. Le Guernic, C. Jacquemot, "Synchronous programming with events and relations : the Signal language and its semantics", Science of Computer Programming, 16, 1991, p. 103-149.
- [4] R. Nikoukhah, "Extensions to Modelica for efficient code generation and separate compilation", in Proc. EOOLT Workshop at ECOOP'07, Berlin, 2007.
- [5] R. Nikoukhah, "Hybrid dynamics in Modelica: Should all events be considered synchronous", in Proc. EOOLT Workshop at ECOOP'07, Berlin, 2007.
- [6] R. Nikoukhah, S. Furic, "Synchronous and Asynchronous Events in Modelica: Proposal for an Improved Hybrid Model", in Proc. Modelica Conference, Bielefeld, 2008.
- [7] P. Fritzson - "Principles of Object-Oriented Modeling and Simulation with Modelica 2.1", Wiley-IEEE Press, 2003.
- [8] S. L. Campbell, Jean-Philippe Chancelier and Ramine Nikoukhah, "Modeling and Simulation in Scilab/Scicos", Springer, 2005.